A common misinterpretation is that — because a POD `type` can be copied around as bytes — it is permissible to interpret any suitably aligned sequence of bytes as an object of that type:

```
void ex5()
{
    struct S { short x, y; } s = { 1, 2 };  // POD type initialized to { 1, 2 }
    int i;
    static_assert(sizeof s == sizeof i, "");  // true on most platforms
    std::memcpy(&i, &s, sizeof s);            // Bug (UB), S is not int.
    assert(((S&)i).y == 2);                   // Maybe?!
}
```

Modifying a **`const`** object in any way has undefined behavior; hence, even a trivially copyable type that contains a nonstatic **`const`** data member is ineligible to be copied via `std::memcpy`:

```
void ex6()
{
    struct S { const int x; int y; } s1 = {3, 4}, s2 = {};  // S is trivial.
    static_assert(std::is_trivially_copyable<S>::value, "");
    std::memcpy(&s2, &s1, sizeof(S));  // Bug (UB), changes the value of const x
    assert(s2.y == 4);                 // Maybe?!
}
```

A base-class object of even POD type is ineligible to be the source or destination of an `std::memcpy`:

```
void ex7()
{
    struct Bx { char c; } bx1 = { 11 }, bx2 = { 22 };  // nonempty POD struct
    struct Dx : Bx { }    dx1 = {    }, dx2 = {    };  // nonempty POD struct

    // Bug (UB), copy from base-class subobject.
    std::memcpy(&bx1, static_cast<Bx*>(&dx2), sizeof(Bx));
    assert(bx1.c == 0);                                  // Maybe?!

    // Bug (UB), copy to base-class subobject.
    std::memcpy(static_cast<Bx*>(&dx1), &bx2, sizeof(Bx));  // Bug, UB
    assert(static_cast<Bx&>(dx1).c == 22);                  // Maybe?!
}
```

Note that if in, say, generic code we were to use `std::memcpy` to copy an empty POD object (e.g., `by2` below) of nonzero size to a base class object of that same type, we might inadvertently clobber the first data member (e.g., `c`) in a derived-class object (e.g., `dy1`) that employs the **empty base optimization**: